Portunes: generating attack scenarios by finding inconsistencies between security policies in the physical, digital and social domain

Trajce Dimkov, Wolter Pieters, and Pieter Hartel
University of Twente, The Netherlands

{trajce.dimkov, wolter.pieters, pieter.hartel}@utwente.nl

Abstract. The security goals of an organization are implemented through security policies, which concern physical security, digital security and security awareness. An insider is aware of these security policies, and might be able to thwart the security goals without violating any policies, by combining physical, digital and social means. This paper presents the Portunes model, a model for describing and analyzing attack scenarios across the three security areas. Portunes formally describes security alignment of an organization and finds attack scenarios by analyzing inconsistencies between policies from the different security areas. For this purpose, the paper defines a language in the tradition of the Klaim family of languages, and uses graph-based algorithms to find attack scenarios that can be described using the defined language.

Keywords: insider threat, physical security, security awareness, security model .

1 Introduction

Organizations loose confidential information through attacks from external actors and *insiders*. Brackney and Anderson [1] define an insider as anyone with access, privilege or knowledge of an information system. The insider threat is a problem recognized as hard and is well established in the security community [2]. The problem is hard for the following reasons: a) insiders have good knowledge of the organization they attack; an insider can work in the same position for years before committing an attack, and has enough time to learn the security policies and their weaknesses; b) an insider has a set of privileges for carrying out everyday tasks, allowing the insider access to various subsystems in the organization; and c) an insider has an established level of trust among his colleagues. By using social engineering on these employees, an insider can further increase his knowledge of the system and gain additional privileges.

A study by Randazzo et. al [3] shows that 87% of the attacks performed by insiders required no technical knowledge and 26% used physical means or

This research is supported by the Sentinels program of the Technology Foundation STW, applied science division of NWO and the technology programme of the Ministry of Economic Affairs under project number TIT.7628.

the account of another employee as part of the attack. This means that protection against insider threat spans through three security areas: physical security, digital security and security awareness of people. Physical security prevents unauthorized access to an asset by access control on buildings, rooms and objects. Digital security is concerned with access control on electronic information systems. Security awareness of employees focuses on resistance to social engineering, and is achieved through education of the employees. To protect each of the security areas requires a different skill set, which is usually delegated to a different group of security experts. Although the policies specified for each security area may be sound when restricted to a single domain, when combined with the policies from the other security domains the combination is not necessarily sound. Thus, a number of actions allowed in a specific security area, when combined, may lead to an attack. Currently, the majority of the security models focus on digital security, and only few consider physical security or security awareness of employees [4]. This paper tackles the problem of presenting and generating attack scenarios in which an insider combines digital, physical and social means to achieve his goal.

The paper addresses the insider problem from a modelling perspective, and in particular in the context of mobility of objects and interactions between people. The contribution of the paper is twofold. Firstly, the paper introduces a formal model, Portunes[1], which combines aspects from the three security areas and presents them in a single formalism. Secondly, the paper uses the model to generate attack scenarios by combining actions which are allowed by the security policies in the organization. To the best of our knowledge, ours is the first approach that generates attacks in which an insider can use a combination of digital, physical and social means to achieve a goal.

The rest of the paper is structured as follows. Related work is presented in section 2. Section 3 formalizes the Portunes ontology, and section 4 presents the syntax and semantics of the language used by Portunes. Section 5 presents a graph-based analysis to assess the security policies presented in Portunes. Section 6 concludes and identifies future work.

2 Related work

The design of Portunes is influenced by several research directions. The model uses the containment relation to define the location of an element, as used by Dragovic and Crowcroft [5], and Scott [6]. Ha et al. [7] associate an insider with a number of credentials and present the world in a graph. As the insider traverses the graph, he can obtain additional credentials which increases the number of actions he can execute. Portunes extends the concept with multiple actors and additionally allows actors to move objects and thus to modify the graph. Probst et al. [8] introduce stratification of elements to distinguish an element according to its physical nature. Portunes defines three distinct layers to reflect the specific properties of the physical and digital domain.

[1] The Roman god of keys

The Portunes language is influenced by the Klaim [9] family of process calculi, or more precisely by the μKlaim [10], OpenKlaim [11] and acKlaim [8] dialects. The language uses security policies, similar to acKlaim, and actions for mobility and embedding of objects (login, logout), similar to OpenKlaim. The Portunes language is simpler than the Klaim family of languages, in the sense that it omits tuple spaces and the actions associated with tuple spaces. Yet, the language is sufficiently expressive to model sophisticated attacks that involve the three security areas.

The majority of approaches that generate attack scenarios use graph-based analysis [12,13]. In these approaches the number of exploits and the number of preconditions in the system that need to be satisfied to execute an exploit are predefined. The paper also uses a graph-based analysis to generate an attack scenario, where the exploits are mapped into actions. However, in a Portunes model, the number of actions and the number of preconditions for an action to happen are not predefined but derived from the graph structure.

3 The ontology of Portunes

This section presents the requirements which Portunes needs to satisfy and the motivation behind some of the decision designs in the model. Based on the requirements, the section defines the ontology of Portunes.

3.1 Requirements and motivation

A security model integrating multiple security areas needs to be expressive enough to present the details of an attack in each security area. In previous work [4] we provide basic requirements for an integrated security model to be expressive enough to present detailed attacks. Briefly , an integrated security model should be able to present the data of interest, the physical objects in which the data resides, the people that manipulate the objects and the interaction between data, physical objects and people. An additional requirement for Portunes is to restrict interactions and states which are not possible in reality. Here are some examples: it is possible to put a laptop in a room, however, putting a room in a laptop is impossible; a person can move only to a neighboring location, while data can move to any location; data can be easily copied, while the reproduction of a computer requires assembling of other objects or materials.

To present the different properties and behavior of elements from physical and digital security, Portunes stratifies a system in three layers: spatial, object and digital. The spatial layers presents the facility of the organization, including rooms, halls and elevators. The object layer is made of elements located in the facility of the organization, such as people, computers and keys. The digital layer presents the data of interest. The division of elements in three distinct layers allows specification of actions that are possible only in a single layer (copying can only happen for digital entities) or between specific layers (a person can move data, but data cannot move a person).

3.2 Formal representation

Portunes abstracts the physical and digital architecture of an organization in a graph. The model stratifies the nodes of the graph in three layers and restricts the edges between layers to reflect reality. A node abstracting a location, such as an elevator or a room, belongs to the spatial layer L and it is termed a spatial node. A node abstracting a physical object, such as a laptop or a person, belongs to the object layer O and it is termed an object node. A node abstracting data, such as an operating system or a file, belongs to the digital layer D. The edges between spatial nodes denote a neighbor relation and all other edges in the model denote a containment relation. The ontology used in Portunes is given in Table 1.

layer	node	edge
spatial	location	neighbors
		contains
object	physical object	contains
		contains
digital	data	contains

Table 1. The ontology of Portunes

The above statements are illustrated in Figure 1 and formalized in the following definition.

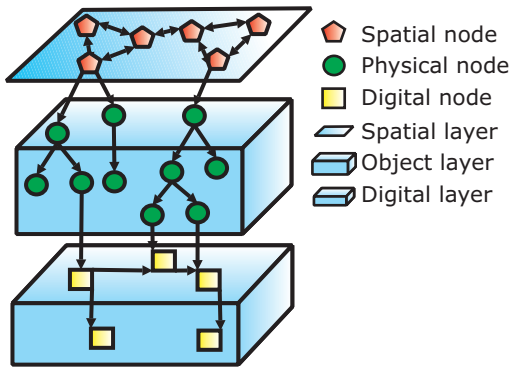


Fig. 1. Graphic presentation of Portunes

Definition 1. *Let* $G = (Node, Edge)$ *be a directed graph and* $\mathcal{D} : Node \rightarrow Layer$ *a function mapping a node to the* $Layer = \{L, O, D\}$. *A tuple* $(G, \mathcal{D})$ *is a Portunes model if it satisfies the following constraints:*

1. *Every object node can have only one parent.*
   $\forall n \in Node : \mathcal{D}(n) = O \rightarrow indegree(n) = 1$
2. *One of the predecessors of an object node must be a spatial node.*
   $\forall n \in Node : \mathcal{D}(n) = O \rightarrow \exists m \in Node : \mathcal{D}(m) = L \wedge \exists \langle m, ..., n \rangle$*; where* $\langle m, ..., n \rangle$ *denotes a finite path from* m *to* n.
3. *There is no edge from an object to a spatial node.*
   $\nexists (n, m) \in Edge : \mathcal{D}(n) = O \wedge \mathcal{D}(m) = L$
4. *There is no edge from a digital to an object node.*
   $\nexists (n, m) \in Edge : \mathcal{D}(n) = D \wedge \mathcal{D}(m) = O$
5. *A spatial and a digital node cannot be connected.*
   $\nexists (n, m) \in Edge : (\mathcal{D}(n) = D \wedge \mathcal{D}(m) = L) \vee (\mathcal{D}(n) = L \wedge \mathcal{D}(m) = D)$
6. *The edges between digital nodes do not generate cycles.*
   $\nexists \langle n, ..., m \rangle : \mathcal{D}(n) = D \wedge n = m$

Rationale: An object node cannot be at more than one place, thus an object node can have only one parent (1). An object node is contained in a known location (2). An object node cannot contain any spatial objects (3) (for example, a laptop cannot contain a room) nor can a digital node contain an object node (4) (for example, a file cannot contain a laptop). A spatial node cannot contain a digital node, and vice versa (5), and a digital node cannot contain itself (6).

4 Portunes language

In the previous section, Definition 1 defines a graph-based model to present the facilities of an organization, the objects in a facility and data of interest. This section looks at the interactions occurring between these elements. The section introduces a language inspired by the Klaim family of languages, which includes actions able to change the graph structure and policies which restrict the actions. The semantics of the language is defined in such a way as to take into account the constraints from Definition 1.

Each node in Portunes is associated with a policy describing which other nodes are allowed to execute actions on it. Physical security deals with access to elements from the spatial and object layer, while digital security deals with access control on elements from the digital layer. Portunes treats people as physical elements that can do physical actions.

The language captures two main properties, *mobility of nodes* and *security awareness* of people. Node mobility allows presenting scenarios such as a key leaving a restricted area, exposing it to the public. Furthermore, while moving through the premises of the organization, insiders *acquire additional privileges*, which increases the number of actions they are capable of. Another property the language captures is security awareness, which is expressed through security policies on people, defining who can give and take elements from a person (whom the person trusts), and who can delegate activities to a person.

4.1 Syntax

As with other members of the Klaim family, the syntax of the Portunes language is divided in three layers: nodes, processes and actions. The Portunes language lacks the tuple spaces present in the Klaim family of languages, and focuses on the connections between nodes. This is because connectivity is the main interest from the perspective of security modeling. Data is thus represented as a node, which is able to move through the graph.

The syntax of the Portunes language is shown in Figure 2. A *node N* consists of a name $l \in \mathcal{L}$, where $\mathcal{L}$ is a finite set of node names, a set of node names $s \in \mathcal{P}(\mathcal{L})$, an access control policy δ and a process P. The set of node names s presents the nodes that are connected to node l in the Portunes model. A *process* P is a composition of actions originating from a single node, called the origin node. An *action a* is a primitive which manipulates the nodes in a net. The *login* and *logout* actions add/remove a node name from s and the *eval* action spawns a process in a target node.

Example: For a node representing a room, $room ::_s^\delta$, the access control policy δ defines the conditions under which other entities can enter or leave the room. The set s contains the names of all nodes that are located in the room. An example of person entering the room is presented through $person :: [login(room)]^{supervisor}$, while $logout(room)$ means that the person exits from the room. To execute these actions, the person must have certain privileges, which are inherited from the node requesting the actions, the origin node *supervisor*.

$N ::=$		Node	$a ::=$		Action
	$l ::_s^\delta P$	Single node		$login(l)$	Login
	$N_1 \parallel N_2$	Net composition		$logout(l)$	Logout
				$eval(P)@l$	Spawning
$P ::=$		Process			
	nil	Null process			
	$[P_1]^{l_{o1}} \mid [P_2]^{l_{o2}}$	Process composition			
	$[a.P]^{l_o}$	Action prefixing			

Fig. 2. Syntax of the Portunes language

A node can execute a set of actions $C = \{ln, lt, e\}$ over another node, where ln is a label for the action *login*, lt for the action *logout* and e for the action *eval*. The access control policy δ is a function specified as $\delta : \mathcal{L} \times \mathcal{L} \times \mathcal{P}(\mathcal{L}) \rightarrow \mathcal{P}(C)$. The first and the second parameter denote identity based access control and location based access control respectively. If the identity or the location does not influence the policy, it is replaced by $\perp$. The third parameter denotes credential based access control, which requires a set of keys to allow an action. Similarly, if a policy is not affected by credentials, the third parameter is empty set.

A security policy can present a situation where: 1) only credentials are needed, such as a door that requires a key $[\perp, \perp, \{key\} \rightarrow ln]$, 2) only the iden-

tity is required, such as a door that requires biometrics $[John, \bot, \emptyset \rightarrow ln]$ or 3) only the location is required, such as data that can be reached only locally $[\bot, office, \emptyset \rightarrow ln]$. The policy supports combination of these attributes, such as a door requiring biometrics and a key $[John, \bot, \{key\} \rightarrow ln]$. The least restrictive policy that can be used is: $\delta(\bot, \bot, \emptyset) \rightarrow \{ln, lt, e\}$.

4.2 Auxiliary functions

Figure 3 defines three auxiliary functions which simplify the operational semantics of the language. The boolean function *grant* takes three parameters. The first parameter defines the node making a request for an action, referred to as the origin node, the second parameter defines the target node and the third parameter is a label of an action. Intuitively, a node can perform an action depending on the credentials it possesses s_o, its identity l_o and its location $parents(l_o)$ which is defined as $\{ l_{po} | \, l_{po} ::^{\delta_{po}}_{s_{po}} \wedge \, l_o \in s_{po} \}$. Formally, an entire node needs to be presented in the grant function, but when it is clear form the context, only the name of the node is used.

The ordering relation $l_t \succ_{ln} l$ states that node l_t can contain node l. The relation can be considered as an access control policy defined by the restrictions Section 3. The relation $l_t >> l$ checks if the ordering relation holds for two nodes from the same layer, and is defined during the instantiation of the model. The relation is transitive but neither symmetric nor reflexive.

$$grant(l_o ::^{\delta_o}_{s_o} P \, , l_t ::^{\delta_t}_{s_t} R \, , a) =$$
$$\exists k_1, k_2, K : a \in \delta_t(k_1, k_2, K) \wedge (k_1 = l_o \vee k_1 = \bot) \wedge (k_2 \in parents(l_o) \vee k_2 = \bot) \wedge (K \subseteq s_o)$$

$$l_t \succ_{ln} l = \begin{cases} true & iff \quad (\mathcal{D}(l_t) = L \wedge \mathcal{D}(l) = O) \vee (\mathcal{D}(l_t) = O \wedge \mathcal{D}(l) = D) \\ l_t >> l & iff \quad \mathcal{D}(l_t) = \mathcal{D}(l) \\ false & otherwise \end{cases}$$

$$l_t \succ_e l = (\mathcal{D}(l) \neq L \neq \mathcal{D}(l_t) \wedge \overline{\mathcal{D}(l) = D \wedge \mathcal{D}(l_t) = O} \wedge (l_t \in s \vee \exists l_p ::^{\delta_p}_{s_p} K : l, l_t \in s_p \vee \mathcal{D}(l_t) = D))$$

Fig. 3. Auxiliary function *grant* and $\succ$ relations

The ordering relation $l_t \succ_e l$ states that node l can take control of node l_t by means of spawning a process. A process originating from a spatial node cannot spawn other processes. An object node can spawn a process to a digital node or another object node, while a digital node can spawn a process only to another digital node. Furthermore, a non-digital node can spawn a process only to a child or sibling. In digital nodes the proximity does not play any role in restricting the spawning of a process. This decision assumes the world is pervasive and two digital nodes can be connected from any location as long as they have the appropriate credentials.

4.3 Operational semantics

Similar to Bettini et al. [11], the semantics of the Portunes language is divided into process semantics and net semantics. The process semantics is given in terms of a labeled transition relation $\xrightarrow{a}$ and describes both the intention of a process to perform an action and the availability of resources in the net. The net semantics given in terms of a transition relation $\Rightarrow$ describes possible net evolutions and relies on the labeled transition $\xrightarrow{a}$ from the process semantics.

$$\frac{l_t \succ_{ln} l \wedge grant(l_o, l_t, ln)}{l ::_s^{\delta} [login(l_t).P]^{l_o} \| \, l_t ::_{s_t}^{\delta_t} Q \xrightarrow{login(l,l_t)} l ::_s^{\delta} [P]^{l_o} \| \, l_t ::_{s_t \cup l}^{\delta_t} Q} \textbf{[login]}$$

$$\frac{grant(l_o, l_t, lt)}{l ::_s^{\delta} [logout(l_t).P]^{l_o} \| \, l_t ::_{s_t \cup l}^{\delta_t} Q \xrightarrow{logout(l,l_t)} l ::_s^{\delta} [P]^{l_o} \| \, l_t ::_{s_t}^{\delta_t} Q} \textbf{[logout]}$$

$$\frac{l_t \succ_e l \wedge grant(l_o, l_t, e)}{l ::_s^{\delta} [eval(Q)@l_t.P]^{l_o} \| \, l_t ::_{s_t}^{\delta_t} R \xrightarrow{eval(l,l_t)} l ::_s^{\delta} [P]^{l_o} \| \, l_t ::_{s_t}^{\delta_t} R | [Q]^{l_o}} \textbf{[eval]}$$

Fig. 4. Process semantics

The process semantics of the language is defined in Figure 4. A node can login to another node if it has sufficient privileges to perform the action and if the node can be contained in the target node **[login]**. Logging out is only dependent on the policies of the target **[logout]**. When a process is spawned, it inherits the name of the origin node from the spawning process **[eval]**.

$$\frac{N \xrightarrow{eval(l,l_t)} N_1}{N \Rightarrow N_1} \textbf{[neteval]}$$

$$\frac{N \xrightarrow{logout(l,l_{t_1})} N_1 \quad N \xrightarrow{login(l,l_{t_2})} N_2 \quad \mathcal{D}(l) = D}{N \Rightarrow N_2} \textbf{[netcopy]}$$

$$\frac{N \xrightarrow{logout(l,l_{t_1})} N_1 \quad N_1 \xrightarrow{login(l,l_{t_2})} N_2 \quad (l_{t_1} \in s_{t_2} \vee l_{t_2} \in s_{t_1})}{N \Rightarrow N_2} \textbf{[netmove]}$$

Fig. 5. Net semantics

The net semantics in Figure 5 defines the possible actions in a Portunes model. Spawning a process is limited solely by the process semantics **[neteval]**. A node can move to a neighboring node if the access control policies of both nodes permit so **[netmove]**. Only digital nodes can replicate, leading to data be-

ing in multiple nodes simultaneously **[netcopy]**. The standard rules for process composition, network composition and structural congruence apply [8].

4.4 Example

The following example shows how a security system comprised of locations, people, objects and data can be modeled by Portunes. Assume that server is located in a restricted area in the facility of an organization. The sensitive data on the server can be accessed only locally and only a few employees have access to the restricted area. An insider with a dongle containing malicious software has a trust relationship with an employee who has access to the restricted area. The insider does not have credentials to enter the secure room. Figure 6(a) abstracts

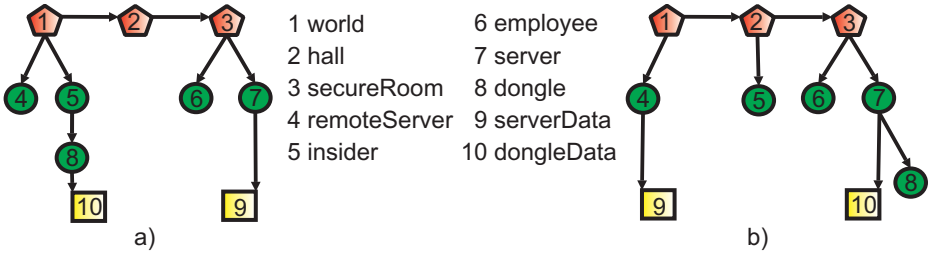


Fig. 6. Portunes model of the system

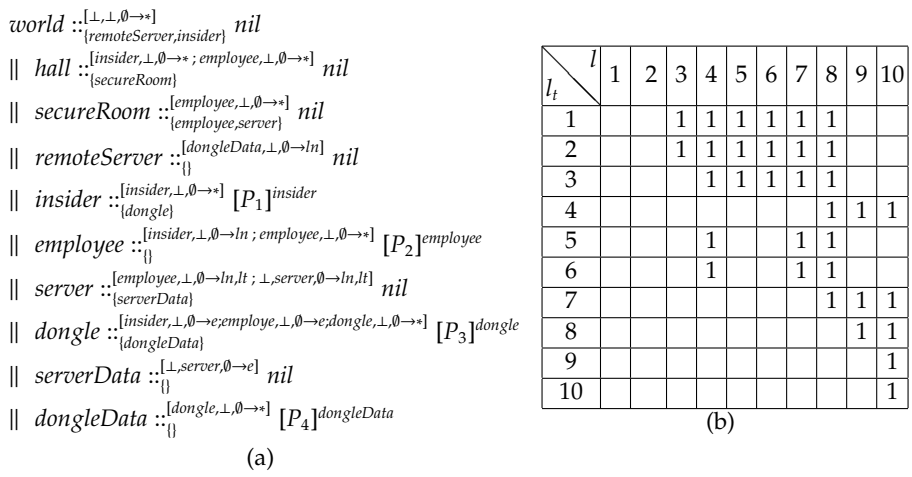
from the example and gives a visual presentation of the Portunes model, and Figure 7(a) gives the formal presentation. The `hall`, `secureRoom` and `world` are spatial nodes, the `serverData` and the `dongleData` are digital nodes and the rest of the elements are object nodes. The $>>$ relation is given in Figure 7(b), where the numbers replace the appropriate node names as shown in Figure 6.

To reduce the number of nodes, the majority of the policies presented here are identity based. For example, the policy $[employee, \bot, \emptyset \rightarrow *]$ on `secureRoom` requires the biometrics of the employee to identify itself when entering and leaving the room. In a less secure environment, the policy can be replaced with $[\bot, \bot, \emptyset \rightarrow lt; \bot, \bot, \{doorkey\} \rightarrow ln]$, meaning that everyone can leave the room, but a person containing a key can enter.

Having abstracted the example system in a Portunes model, the following question arises: can the data reach the remote server without violating any policy? If possible, for which actions in P_1, P_2, P_3 and P_4 can the data move to a remote server?

5 Analysis of a Portunes model

This section provides an analysis on a Portunes model, which consists of three algorithms and returns an attack scenario. The algorithms used in the analysis



l_t \ l	1	2	3	4	5	6	7	8	9	10
1			1	1	1	1	1	1		
2			1	1	1	1	1	1		
3				1	1	1	1	1		
4								1	1	1
5				1			1	1		
6				1			1	1		
7								1	1	1
8									1	1
9										1
10										1

Fig. 7. The example in Portunes language and the >> relation

implement the rules from Figure 4 and Figure 5 from the language semantics to judge whether an action is possible, such that the attack scenario does not violate any existing policy.

5.1 Algorithms

The first two algorithms are based on the algorithms from Ammann et al. [13], consisting of a forward marking stage and a backward attack finding stage. The basic items to evaluate are *attributes* and *actions*[2]. A model instance starts with a particular set of attributes, of which some are initially satisfied. An action is an event that increases the number of attributes that are satisfied in the model. In case of Portunes, attributes are a) whether an entity is logged into another entity and b) whether an entity has a process originating from another entity due to an *eval*. Actions are the net actions **[netcopy]** and **[neteval]** from the net semantics of the Portunes language. The first two algorithms use the *monotonicity* assumption, which states that the precondition of a given action is never invalidated by the successful execution of another action. In the physical world, this assumption means a person able to enter a room can never loose this ability, presenting the most pessimistic scenario where the adversary never looses a capability. Since **[netmove]** removes an edge in the graph, it invalidates an attribute and thus the monotonicity rule. Thus, **[netmove]** is identified with the **[netcopy]** action. The third algorithms shows how the monotonicity rule can be lifted and generate real life scenarios.

The first algorithm, `generateActionTemplates` (called `markAttributes` in [13]) creates a forward analysis of what can go where. The algorithm serves two purposes, to show which locations an element can reach and to generate

[2] Termed exploits in Ammann et al. [13]

a set of action templates that might occur in the Portunes model. The main difference from `markAttributes` in [13] is that the instantiation of the actions and the number of instantiations is not known in advance, but is calculated by the algorithm.

```
   name : generateActionTemplates
   input : Set of initially satisfied attributes A
   input : Set of nodes with policies N
   output: Set of unique action templates actionT
   type : tuple action = ⟨node, node, node, node⟩
   type : tuple template = ⟨action, set of attributes, atribute, integer⟩
 1 template actionT = ∅
 2 int iteration = 0
 3 repeat
 4     template newActT = ∅
 5     foreach node n in N do
 6     foreach node origin in origins(n) do
 7     foreach node parentOfNode in parents(n) do
 8     foreach node target in N do
 9         if canCopy(origin, n, parentOfNode, target, precondition) then
10             action a = ⟨n, origin, target, parentOfNode⟩
11             template s = ⟨a, precondition, postcondition, iteration⟩
12             if s ∉ actionT then
13                 newActT = newActT ∪ s
14                 add the postcondition of a in the attribute table A
15         if canEval(origin, n, parentOfNode, target, precondition) then
16             action a = ⟨n, origin, target, ⊥⟩
17             template s = ⟨a, preconditions, postcondition, iteration⟩
18             if s ∉ actionT then
19                 newActT = newActT ∪ s
20                 add the postcondition of a in the attribute table A
21     iteration++
22     actionT = actionT ∪ newActT
23 until newActT = ∅
```

Algorithm 1: Generate action templates

An action template generated by the algorithm is composed of an action, the precondition required for the action to execute, the postcondition from the execution of the action and the earliest iteration when the execution is possible. The precondition and the postcondition are defined in terms of attributes of the model. In `generateActionTemplates` the *postcondition* of **[netcopy]** is the satisfaction of the attribute representing that node *n* is logged into node *target*. The postcondition of **[neteval]** is the satisfaction of the attribute representing

that node n has spawned a process at node *target* (and can thus be the originator of a future action by *target*).

In each iteration, the algorithm checks if a node can be copied to a target node or spawn a process from the node to a target node. The function `origin` returns all processes resigning in a node, while the function `parents` returns all the parents of a node. The function `canCopy` evaluates according to the premises presented in Figure 5. Depending on the access control policy, `canCopy` might evaluate to `true` because of the location of the origin, the identity of the origin or the set of nodes the origin contains. For this reason, `canCopy` returns a minimal set of attributes *precondition* with which the action can be executed. Every new action template is added to the set of possible action templates together with the iteration the action is discovered at. Since the postcondition of an action can enable a precondition of another action, the algorithm iterates until all of the nodes cannot perform any new unique actions.

In a Portunes model composed of N nodes, a node can contain $N-1$ other nodes and processes from N origin nodes. Therefore, the model has $N \times (N-1) \approx N^2$ attributes. Because of the monotonicity assumption, the number of satisfied attributes can only increase, lading to a maximum of $O(|N^2|)$ iterations in the algorithm. We assume the policy restrictions on each node have a constant bound concerning the number of nodes, which makes the output of the functions `origin` and `parents` constant. Thus, each iteration is of complexity $O(|N^2|)$, leading to worst case complexity of the `generateActionTemplates` algorithm of $O(|N^4|)$.

The resulting set of action templates allows us to ask a question *how* an asset can reach a location. It is now possible to generate the actions that lead to a particular goal by backtracking from the goal to the initial situation, following the post- and preconditions of the action templates. The second algorithm, `findPartialAttack` generates such an attack scenario. The resulting attack scenario is monotonic since it does not contain any cyclic movement of the nodes. An example of a cyclic movement is an insider going from a hall to a room to obtain a key, and returning to the hall to continue with the attack.

The `findPartialAttack` algorithm uses a goal, the action templates generated from the `generateActionTemplates` algorithm and a state of a Portunes model, and returns a set of action templates leading to a partial attack. The variable *goal* is an attribute, *partialResult* is a set of all used action templates, *iteration* is the iteration in which the action is found, *satisfiedAttributes* is a set of attributes that are satisfied by the execution of actions in *partialResult* and *actionT* is a set of all action templates. The function `find` starts from *goal* and finds an action template whose postcondition satisfies the goal. For each attribute in a precondition, `find` recursively searches for an action template occurring in a previous iteration whose postcondition satisfies the attribute of interest. Previously used action templates are omitted, as well as any attributes satisfied by previously used action templates. The resulting set of action templates *partialResult* contains the set of actions, which ordered by iteration presents a partial attack scenario.

```
   name : findPartialAttack
   input : Set of action templates actionT
   input : Attribute goal
   output: Set of action templates leading to the goal partialResult
 1 set of attributes satisfiedAttributes = ∅
 2 set of templates partialResult = ∅
 3 find(iteration, goal, satisfiedAttributes, partialResult)
 4 begin
 5     find a template s ∈ actionT such that s.postcondition = goal and
       s.iteration ≤ iteration and s ∉ partialResult
 6     if there is no such template then return error
 7     partialResult = partialResult ∪ s
 8     foreach attribute p in s.precondition do
 9         if p ∉ satisfiedAttributes then
10             satisfiedAttributes = satisfiedAttributes ∪ p
11             find(s.iteration, p, satisfiedAttributes, partialResult)
12 end
```

Algorithm 2: Generate a monotonic attack scenario

In the worst-case scenario, the function `find` is executed for each attribute in the model. Assuming that the number of pre- and postconditions per action does not depend on the number of nodes N, the complexity of the algorithm is $O(|N^2|)$.

Because of the monotonicity assumption, the set of action templates generated by `findPartialAttack` does not contain actions generating cyclic movement of the elements. The `simulate` algorithm does not use the monotonicity assumption and adds additional action templates in the monotonic attack scenario to include scenarios where an object must be returned to a previously visited place to complete the attack. Contrary to the first algorithm, in `simulate` the net action **[netmove]** removes the edge between the parent and the node where the action is executed.

The `simulate` algorithm uses a set of action templates *partialResult*, which are generated by `findPartialAttack` and a set of attributes satisfied from the initial state of the Portunes model, and returns a list of action templates which represent an attack scenario. The algorithm takes an action template from the earliest iteration and checks if its precondition is met. If all attributes in the precondition are satisfied, the algorithm executes the action from the template and updates the attributes with the postcondition from the action template. If a precondition is not satisfied because it is invalidated by another action, the algorithm generates a new partial scenario which tries to satisfy the precondition that and continues the simulation. The variable *result* is a list containing an attack scenario which might include cyclic movement and is semantically valid when translated into the Portunes language. These can then be translated to the Portunes language by grouping the actions by origin node.

The maximum number of actions is $O(|N^2|)$, and for each action, the algorithm might need to generate a cyclic movement. From the previous algorithm, the complexity of generating a partial scenario is $O(|N^2|)$. Assuming the existence of only simple cycles, the worst case complexity of `simulate` is $O(|N^4|)$.

```
   name : simulate
   input : Set of initially satisfied attributes A
   input : Set of action templates representing a partial attack scenario
           partialResult
   output: List of action templates representing an attack scenario result
 1 begin
 2     while actionT ≠ ∅ do
 3         let a be an action template with the smallest iteration number in
           partialResult
 4         let S be the set of attributes in the precondition not satisfied in A
 5         if S == ∅ then
 6             result.Append(a)
 7             partialResult = partialResult \ a
 8             change attributes in A based on nonmonotonic postcondition
 9         else
10             set of templates addActT = ∅
11             find (a.iteration, a.postcondition, A, addActT)
12             partialResult = partialResult ∪ addActT
13 end
```

Algorithm 3: Generate an attack scenario

5.2 Example (continued)

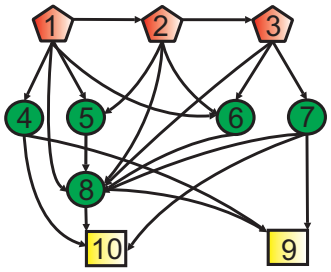
This section continues the example from Section 4.4 and shows how the analysis from the previous section finds a possible attack scenario.

The first algorithm generates a set of all possible action templates that can be done by the insider, the employee, the dongle and the malicious software. An example of an action template with a *netmove* action is presented below. To move the *serverData* from *server* to *remoteServer*, two attributes need to be satisfied: the *server* needs to contain the *serverData* and the *serverData* contains process originating from *dongleData*. As a result of the execution of the action, two attributes change: *server* does not contain the *serverData* and the *remoteServer* now contains the *serverData*. This action is discovered during the eight iteration of the `generateActionTemplates` algorithm.

After the execution of the algorithm, the Portunes model is transformed in a graph including all allowed edges between the nodes (Figure 8(b)). The edge from *remoteServer* (4) to *serverData* (9) means that *remoteServer* at one point contains *serverData*.

action:
serverData :: [*logout*(*server*).*login*(*remoteServer*)]^{dongleData}
precondition:
server contains *serverData*
data contains process originating from *dongleData*
postcondition:
server does not contain *serverData*
remoteServer contains *serverData*
iteration: 8

(a)



(b)

Fig. 8. somethingelse

The `findPartialAttack` algorithm uses the generated set of action templates, the initial Portunes model and the goal: *data* in *remoteServer* to generate a partial attack scenario. After merging the actions with same origin and ordering them by iteration number, the processes are:

P_1=[*logout*(*world*).*login*(*hall*). (a)
eval(*logout*(*insider*).*login*(*hall*).*logout*(*hall*).*login*(*employee*))@*dongle*]^{insider} (b)
P_2=[*logout*(*secureRoom*).*login*(*hall*). (c)
eval(*logout*(*employee*).*login*(*secureRoom*). (d)
logout(*secureRoom*).*login*(*server*))@*dongle*]^{employee} (e)
P_3=[*eval*(*logout*(*dongle*).*login*(*server*))@*dongleData*]^{dongle} (f)
P_4=[*eval*(*logout*(*server*).*login*(*remoteServer*))@*serverData*]^{dongledata} (g)

One interpretation of the actions is the following. The insider (P_1) goes in the hall and waits for the employee (process P_1 is then at point a). When the employee (P_2) arrives in the hall (P_2 at c), the insider gives him the dongle containing malicious software, which the employee accepts (P_1 at b). Later, the employee plugs the dongle in the secure server (P_2 at e) using its own credentials and the server gives the dongle (P_3) access to the local data. When the malicious software (P_4) reaches the server, it sends all the data to the remote server. The above actions closely resemble the road apple attack [14] with a dongle automatically running when attached to a computer [15].

The resulting scenario is still not complete. When the dongle reaches the employee, the dongle tries to move to the secureRoom, although the employee is located in the hall at that moment. Thus, the part of the attack scenario where the employee returns back to the secure room is missing. After running the `simulate` algorithm, this action is also included, and the employee process gets

the following form:

$$P_2 = [logout(secureRoom).login(hall).eval(logout(employee).login(secureRoom).\ logout(secureRoom).login(server))@dongle.\mathbf{logout}(\mathbf{hall}).\mathbf{login}(\mathbf{secureRoom})]^{employee}$$

After running the Portunes program, the final state of the Portunes model is given at Figure 6(b).

In the above example, the analysis combines physical, digital and social aspects of security. From the example, one can observe that enforcing a policy which forbids a server to accept remote connections is useless if there is no physical security policy regulating which person can physically reach the server. Additional organizational policy should address dongle use among employees.

6 Conclusion

The paper presents a formalism that integrates several aspects from the real world that are of interest to the physical security, digital security and the security awareness of people in an organization. These aspects are: 1) physical properties of elements, 2) mobility of objects and data, 3) identity, credentials and location based access control and 4) trust and delegation between people. This formalism is used as a foundation for an analysis which uses the policies in these security areas and generates possible attack scenarios that combine physical, digital and social means to achieve an attack.

The main improvements of Portunes upon existing work are:

1. expressing mobility of all objects, not just keys;
2. expressing interaction of the attacker with other people, not just objects and data;
3. adaptation of graph-based vulnerability analysis on facilities and physical objects, not just on computer networks.

In future, we plan to focus in two areas. Firstly, we plan to perform case studies to validate the approach and its usability, and secondly, to improve the scalability of the algorithms presented in the analysis.

References

1. *Understanding the Insider Threat: Proceedings of a March 2004 Workshop*. RAND Corporation, 2004.
2. INFOSEC Research Council. Hard problem list, November 2005.
3. M.R. Randazzo, M. Keeney, E. Kowalski, D. Cappelli, and A. Moore. Insider threat study: Illicit cyber activity in the banking and finance sector. 2004.
4. T. Dimkov, Q. Tang, and P. H. Hartel. On the inability of existing security models to cope with data mobility in dynamic organizations. In *Proceedings of the Workshop on Modeling Security*, 2008. CEUR Workshop Proceedings.

5. B. Dragovic and J. Crowcroft. Containment: from context awareness to contextual effects awareness. In *Proceedings of 2nd Inernational Workshop on Software Aspects of Context*, 2005. CEUR Workshop Proceedings.
6. D.J. Scott. *Abstracting Application-Level Security Policy for Ubiquitous Computing*. PhD thesis, University of Cambridge, 2004.
7. D. Ha, S. Upadhyaya, H. Ngo, S. Pramanik, and R. Chinchani. Insider threat analysis using information-centric modeling. In *IFIP International Conference on Digital Forensics*, pages 55–73. Springer, 2007.
8. C. W. Probst, R. R. Hansen, and F. Nielson. Where can an insider attack? In *Workshop on Formal Aspects in Security and Trust (FAST 2006)*, pages 127–142. Springer, 2006.
9. R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on software engineering*, 24(5):315–330, May 1998.
10. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. 2719:119–132, 2003.
11. L. Bettini, M. Loreti, and R. Pugliese. An infrastructure language for open nets. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 373–377, 2002.
12. L. Wang and S. Jajodia. An approach to preventing, correlating, and predicting multi-step network attacks. *Intrusion Detection Systems*, 2008.
13. P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 217–224. ACM, 2002.
14. S. Stasiukonis. Social engineering the usb way. `http://www.darkreading.com/document.asp?doc_id=95556`, 2006.
15. M. AlZarouni. The reality of risks from consented use of usb devices. In C. Valli and A. Woodward, editors, *Proceedings of the 4th Australian Information Security Conference*, pages 5–15, 2006.